Q22：括号生成 N对括号：可以生成不同的有效括号种类数。种类数，一般都是回溯，才能查到所有情况了。

1. 任何时候，左括号个数，必须大于等于 右括号个数。即是合法的。所以可以用回溯法。并设定left < right 的条件。
2. 每一层可以加一个括号。可以加左，也可以加右，加右要满足left < right
3. 直到 left = 0,right = 0，结束。

Q84：柱状图中最大的矩形 **重点**

1. **存下标的单调栈 枚举矩形的高度**
2. 思想： 关于这种数组中找最大矩形， 的题。 可以明确的是，矩形的高，一定是数组中某元素的值。
3. 所以： 只要我们枚举每一个元素作为高，求其最大宽度即可。
4. 所以： 这题中， 当我们读到一个元素，他小于之前的元素， 那么之前的元素作为高的矩形就一定已经确定了。因为水桶效应。
5. 若nums[i] < nums[stack[top]] 那么以stack[top]为左边界的矩形就确定了，说明 stack[top] 到 i之间的元素都大于nums[stack[top]]，否则stack[top]一定会出栈。

   假设有 23547 当i到4的时候栈中是235，发现比5低， 那么以5作为高的矩形就确定了。所以栈中变为 234
   假设有 235647 当i到4的时候栈中是2356， 发现比6低，那么以6作为高的矩形就是他本身。 然后栈变为 235， 4<5,那么以5作为高的矩形也确定了。 他的右侧边界就是 i-1.也就是6的位置。56组成以5作为高的矩形。
   假设有 212...后序都大于2 当遇到1的是偶栈中是 2， 1<2 那么以2作为高的矩形就是他本身。 然后栈变为1， 2再入栈。。。。
   最终假设栈中只保留了 12 那么以2作为高的矩形的右侧边界就是数组边界，因为数组右侧都大于2。
   还有一个题也是类似： 即可以装数组元素作为边沿，找最大矩形可以装多少水。 也是枚举水桶边。 让每一个元素作为边，找最大宽度。

Q85：最大矩形：01矩阵中： 由1填充的最大矩形。

1. 乍一看是动态规划
2. 其实是单调栈。 而且就是Q84的变行
3. 把第一行作为初试高度heights[] ,然后求若只有第一行的最大矩形。
4. 然后第二行，若第二行是1，那么该列的高度就可以加1. 若是0，那么该列的高度就改为0. 每次都通过Q84题来求解heights的最大矩形。
5. 然后每加入一行，就更新一次heights[]的高度。

```
    for(int i = 0; i < lenr; i++){
        for(int j = 0; j < lenc; j++){
            heights[j] = matrix[i][j] == '1' ? heights[j]+1 : 0;
        }
        int a = largestRectangleArea(heights); //
        ans = Math.max(ans, a);
    }

    largestRectangleArea(heights) : Q84的函数。
```

Q155 : 最小栈： push,pop,top getMin 要求可以在常数时间返回栈中最小值。

0. 要求：首先，因为有push,pop,top操作，我们必须保存原始的入栈顺序。又因为有 getMin，我们需要维护当前栈最小值。

1. 方法1：

    1. 使用两个栈，一个按入栈顺序入栈即可，保留所有值
    2. 再使用一个单调递减栈，当栈为空或者栈顶大于等于x时，x可以入栈。若x大于栈顶，则x不入栈。因为x之前比有更小的值了。
    3. 即使有出栈，也是x先出栈。
    4. 出栈时，只有当x == peek()时，单调栈才出栈一个。
    5. 注意这里：相等时，x也要入栈单调栈，比如0,2,0 最后一个就必须入栈，如果不入栈，那么单调栈只有一个元素0，当普通栈
    6. 弹出第二个0时，单调栈也会弹出一个0，导致单调栈空了，可实际上，第一个0应该在单调栈中。

2. 方法2 ：

    1. 使用自建的Node(x,min1)头插法来表示一个栈。min表示当前结点到栈底的最小值。
    2. 如果来一个新的值y, 那么 插入结点应该是 Node(y,min(y,min1)) getMin直接返回head.min即可。
    3. 弹出时，该min随同y也会消失。
    4. 该方法仍然属于使用了额外的空间，因为对于每一个结点，我们都多使用了一个数据 min.

Q232 用栈实现队列： 双栈实现队列

Q316 去除重复字母 s = "cbacdcbc" ans = acdb 返回结果必须是字典序最小的选择。

1. 方法1: 当num < stack[top]时，

    1. 首先使用 freq[26]对s进行字符频率统计。然后设置vis[26]数组，表示某个字符是否在栈中，入栈则为true,出栈则为false。栈中不允许出现两个相同的字符。
    2. 然后使用单调递增栈。若某未入栈元素>栈顶，就直接入栈。否则就要进行单调栈的维护，把栈顶出栈。出栈还要多一个条件，因为栈中只有一个元素a， 要删除他的条件就是freq[i]>0.栈中最少要用一个，这个时候可以不是单增的。

2. 方法2： error

1. 使用自建双向链表Node作为栈。然后使用HashMap<Integer,Node>标记栈中对应值的结点。
2. 若num[i]已经在栈中，则用map找到，然后num[i]和map.get(i).right.val比较。若num[i]>map.get(i).right.val,则删除链表中的num,然后添加到链表尾部。若链表中没有，直接添加到尾部。 error : "bcabc" 输出 : bac 答案 = abc

3. 单调递增栈确保了不会出现方法2的这种情况的发生. 实质是,仅仅通过num[i]>map.get(i).right.val后边一个值不能判断已经存在的值是否应该删除.

Q402 : 移掉K位数字 可以移掉任意位的数字，如 10200，可以移除1，剩下 0200， 只是输出是把前导0省略，输出 200

1. 删除k位,使得剩下的最小.
2. 特点，1432219： 如果要移除 2位。那么就是移除 43， 移除3位就是移除 432 移除4位就是 4322.
3. 什么规律？从左到右找第一个，当cur位大于(严格大于)cur+1位时，删除cur位。就一定合算的。

```
        使用top=0表示栈顶。每确定一个不删，就入栈top++。
        确保top>=0  k>0

        for(int i = 1; i < num.length(); i++){
            char c = num.charAt(i);
            while( k > 0 && top >= 0 &&  c < stack[top]){ // top 可能为 -1
                top--;
                k--;
            }
            stack[++top] = c;  // 删除以后再入栈c
        }
        top -= k;  // 当i==len时出for循环。此时k可能仍然>0.
        stack[++top] = c;  // 删除以后再入栈c

        假如输入: 1432239 那么删除3位就是 432  删除3位还剩 1239 ,由于后边是递增序列。必然不能删除第4位了。
        于是出for循环时，k = 1 >0  还需要删除一位。那么删除谁呢?
        递增序列，必然删除最大的。即后边的k位。 直接把top-=k即可。
        于是输出就是 0~top的字符。
```

Q456 : 132 模式

1. 维持一个单调递减的栈。 数组逆序入栈.
2. 每次我们都把栈顶当作'3',维持单调递减栈，
3. 入栈'3'时(大于栈顶)，会导致目前的栈顶'3'出栈。 于是，栈顶变为新的'3'， 出栈的'3'中最大的就是
4. 若nums[i] < '2'则说明出现了 132

```
 int max_2 = Integer.MIN_VALUE;
 // 初试给 max_2最小值.这样当 max_2 等于 最小值时,不会出现 min_1 < max_2,直到
max_2更新后.
 stack.push(nums[len-1]);
 for(int i = len-2; i >= 0; i--){
     if(nums[i] < max_2) return true;
     while(!stack.isEmpty() && nums[i] > stack.peek()){
         max_2 = Math.max(stack.pop(), max_2);
     }
     if(nums[i] > max_2)
         stack.push(nums[i]);
 }
```

Q503：下一个更大元素 II 1,2,1 -》 [2,-1,2]

1. 使用入栈 数组下标的形式，维护单调递减栈， 每次遇到需要出栈的时候，即找到了这些出栈元素的更大值。